# Introduction to Java Threads

Process => a program in execution is the process => it has its own memory

Application => Process => Application can be made up of more than one process which cooperate with each other => Client Server Application => Client is a process => Server is another process and there’s an intercommunication between process => Socket programming

Thread => within the process => each thread shares the resources and memory of the process => light weighted process => W => but since they will be accessing the shared resources and works in the shared memory => can leads to concurrency issues

Thread => is the way to achieve the concurrent programming

1. Pseudo concurrency => single core => that’s means only one thread can get executed in any point in time => how come it is a concurrent => time slicing

2. Real concurrency => using multi core with time slicing or using multiple processor

When you create an application by default it is a STA => Single Threaded Apartment => but you can add more threads into your application => Multi-Threaded Apartment.

In java we can create multi-threaded application in two ways:

1. Inheriting (extending) Thread class

2. Implementing Runnable interface – And semantically if you extending a class that means you're extending its capabilities. We're not extending Thread class behaviour

Q: Just a thought, why give the extending option at all then?

A: A user must extend thread class only if it wants to override the other methods in Thread class. If you only want to specialize run method then implementing Runnable is a better option.

Two methods to create multi-threaded programmes:

1. Sub-classing the **Thread** class
2. Implementing the **Runnable** interface

Thread 🡪 lightweight process as:

* runs within the context of a full-blown program
* takes advantage of the resources allocated for that program
* accesses the program’s environment
* must have its own resources within a running program, e.g. its own *execution stack*, its own *program counter*.

Both Processes and Threads provide an execution env.

Threads exists within processes and share its resources like memory, files.

Problems 🡪 Interference & Deadlocks

Main thread has the ability to create more threads

Thread 🡺 single sequential flow of control within a process

## Creating a Thread

1. by *sub-classing* the Thread class & *overriding* its run() method
2. creating a Thread with “a Runnable object as its *target*

i.e. use an object that implements the Runnable interface and provide an implementation of the run() method.

## Thread Body

All of the action takes place in the thread’s body (thread’s run() method). How the run() method is defined that define what’s happening in the thread

## Thread State

A *thread’s state* indicates what the thread is doing & what it is capable of doing at that time of its life: *running*, *not running*, *waiting*, *sleeping* or *dead*.

## Thread Priority

*Priority* indicates to the Java thread scheduler when this thread *should run* in relation to all of the other threads.

Thread priority is between 1 to 10

1is low priority

5 is mid priority 🡺 default

10 is high priority

## Daemon Threads

are those that provide a *service* for other threads in the system. Any Java thread can be a daemon thread.

Threads running in the background.

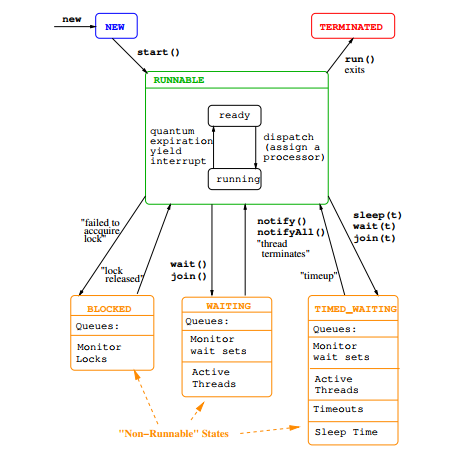
## Thread Group

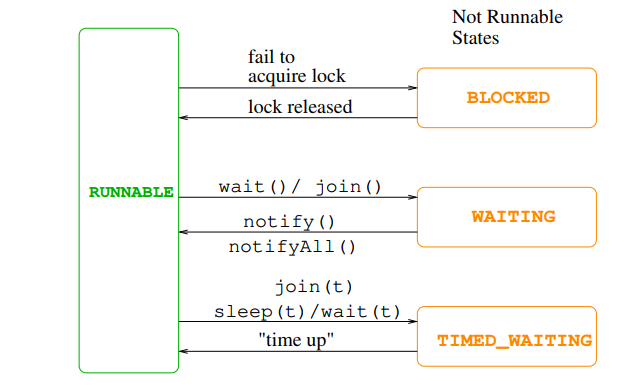
All threads belong to a *“thread group”*. *ThreadGroup*, is a *java.lang* class, which defines and implements the capabilities of a group of related threads.

## Java Virtual Machine

* can support many threads at once
* independently execute code that operates on values & objects residing in shared main memory
* communication is done through shared variables i.e. the variables kept in the main memory & shared by all threads
* main memory contains master copy of every variable, and every thread has its own working memory where it keeps its own working copy of variables eh: a, b, c
* local copies of variables are used for transfer to & from main memory
* threads are executed by Thread Execution Engine
* Its impossible for one thread to attain parameters/ local variables of another thread
* Main memory contains locks & there is one lock associated with each object
* Threads may compete to acquire locks, but will eventually release it

# Thread Life Cycle





What are the various state at which java thread exists and how the transition occurs (How it go from one state to another state and from a state what are possible action can be, performed on the thread),

When there is no interweaved communication between the producer and consumer will leads to following problem:

1. Data loss

2. Double consumption

Wait) => when the wait() method is called the thread enters the WAITING state to come out of the WAITING state another thread needs to call the notify() or notifyAll() to bring back to Runnable thread => threads will be on WAITING state on the monitor object in our example there will be queue on the SimpleMilbox object

‘wait () method can only be called inside synchronized block or method if not IllegalMonitorException will be raised

When a thread calls the join() on another thread the thread which called join() method will relinquishes the resources and goes into WAITING state until the thread on which the join() method called completes the execution

Main thread and there is another thread that is created by the main thread called t1.

If the main thread calls the join) method in the tl (t1 Join) => main thread goes into WAITING state until the {1 complete the execution.

Join(ms) => either waits for the time to expire or waits for the thread on which join(ms) is called to terminate (run method to complete the execution) whichever the earliest

Depreciated method in Thread class:

Stop() => which is used to stop a thread. => problem with this when the stop method is invoked a ThreadDeath object is thrown and the thread will stop at the point of receiving the ThreadDeath object. main thread throws a ThreadDeath object on a thread called Thread-01 what happens at the point it receives the ThreadDeath object only the Thread-01 will die and this happens asynchronously => Thread-01 might be in the critical section and the ThreadDeath object i received and it will die in the middle of critical section => as result the object on which the thread was operating will be left in the inconsistent state which is known as a damaged object.

Thread will terminate naturally when the run method completes the execution but what if the run{) method has an infinite loop (Timer Applet example)

1. stop, suspend, resume and destroy methods of java Thread is depreciated

2. Why stop method is depreciated => stop() throws a ThreadDeath object and when the Thread receives that ThreadDeath object its getting killed (asynchronously) => danger part is while the ThreadDeath object's being received the thread might be in the critical section

So the object on which thread was, working might be left in the inconsistent state => otherwise the monitor object might be damaged

3. Suspend method is also depreciated => due to the reason that it might move from RUNNABLE to NOT RUNNABLE while holding lock on the resources (monitor object) => deadlock

4. resume method is used to move the thread on which suspend method is invoked from NOT RUNNABLE to RUNNABLE because the suspend method is depreciated no point of using resume method as result that also depreciated

5. destroy() method of the Thread class, works in similar fashion to suspend() => only difference is it will move the Thread permanently from RUNNABLE to NOT RUNNABLE

# Thread Scheduling

Synchronization: is required to *coordinate* the read & write actions of these threads to stop them interfering with each other & ensuring *“data integrity”*.

Starvation: occurs when one or more of the threads is **blocked** from accessing the file & **cannot make progress**.

Deadlock: is the ultimate form of starvation & occurs when two or more threads are **waiting on a condition that cannot be satisfied**.

Thread Scheduling:

Concurrent Programming can be categorized under two categories based on the number of cores that a machine has:

1. Real concurrent programming => where there are multi core processor where each thread will get executed truly simultaneously in different core or multiple processor each of the thread can executed in different processor in that case also itis truly concurrent

2. Pseudo (fake) concurrent programming => there are multiple threads wanted to get executed but there’s only a single core => that means at any given time only one thread can get executed => then how can you called itis a concurrent programming?

This where time slicing (Quantum) comes into play => each of the thread get a time slice (Quantum) execute as a result it looks like they are executing concurrently through in realty only one threads runs at any given time

3. Why do we need thread scheduling? Because there are multiple threads and someone has to pick one of them and give them the (time slice) to execute

4. JVM uses something called => Fixed priority pre-emptive thread scheduling => Thread with highest priority in the READY state will be allocated quantum of the processor to execute when the processor falls vacant —> the highest priority thread will be moved from READY to RUNNING state in the RUNNABLE

‘What if low priority thread is now in the RUNNING state and higher priority thread centres the READY state. What you think will happen? => the higher priority thread will pre-empt the lower priority thread => now the lower priority thread will be pre-empted -> kicked out of the processor => goes from RUNNING to READY while the higher priority thread will take up the processor => goes from READY to RUNNING

Based on the above fact we call the scheduler uses a scheduling algorithm of Fixed priority pre-emptive scheduling

What you think will be the consequences of higher priority thread pre-emps the lower priority thread and getting executed? **Starvation** => how the starvation is being avoided” The thread scheduler uses a mechanism called aging -> based on the time period for which the thread is put into **WAITING** period its priority is elevated by the scheduler $o eventually after wasting for some time the thread with lower priority will 1st the chance to execute “> once execution is over the original priority will be taken

‘What if two threads with same priority enters the READY state who will be moved to RUNNING state? They will follow the round robin basis and execute one after the other taking time slice

Thread priority => priority is an attribute or instance variable of Thread object => each thread has a priority and its value can range from | to 10

Thread.MIN\_PRIORITY = 1

Thread.NORM\_PRIORITY = 5 (that is the default value of the priority attribute of the thread)

Any thread that is created inherits the valve for the priority attribute from the thread creates it therefore as all the threads are created under the main method, they get the priority 5

Thread.MAX\_PRIORITY = 10

Priority attribute can be accessed and mutated by getter and setter:

setPriorty(): int > returns the priority and value between 1 to 10

‘setPriority(in priority): void => sets the priority attribute to the parameter

# Thread Groups

You can create thread groups and place thread and thread groups into it.

Thread group if the parent thread group is not specified at the time creating it belong to the thread group of the thread creating it (Thread which is creating the thread group will place the thread group under the thread group to which the thread belongs to)

*public ThreadGroup (String name)*

3. Thread if created using one of the following constructors then the thread is placed in

the thread group to which the creating thread belongs to:

***Thread ()*** *=>Allocates a new Thread object.*

***Thread (Runnable target)*** *=>Allocates a new Thread object.*

***Thread(Runnable target, String name)*** *=>Allocates a new Thread object.*

***Thread (String name)*** *=>Allocates a new Thread object.*

4. However when you are creating a thread you can specify the parent thread group

using the following constructor:

*public ThreadGroup(ThreadGroup parent, String name)*

# Monitor and Synchronization

A *monitor* consists of: A collection of declarations of *permanent variables*. They are used to represent & indicate the *state of the resource*. A collection of *procedure & function* declarations.

|. Single slot monitor using implicit lock => synchronized keyword and wait() and

notifyAl() method => SimpleMiailBox

Multislot monitor using implicit lock => MultiSlotMailBoximpl

Muttislot monitor using explicit lock => => Lock interface and Reentranant

## Reader writer problem

Write operation => 0 <= no of writer <= 1 simultaneous write operation is allowed but when

there is a write operation going on there cannot any other read or write operation

Read operation => 0 <= no of reader <= n

simultaneous read operations are allowed but when read operation is taking place no write operation

\* no of other read operation is allowed i.e multiple simultaneous read operations are allowed

Can we do this using synchronized? No not possible

We cannot use synchronized black or method because the movement we enter a synchronized block cannot do another read or write why? A lock is established (exclusive lock is established)

# Semaphore

Semaphore there are two types of semaphore

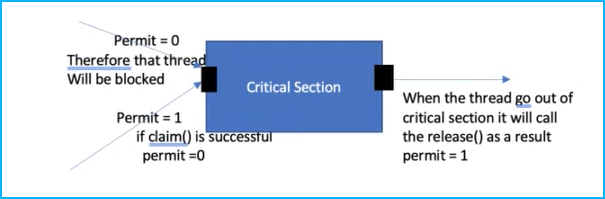
1. Binary Semaphore => Mutex

2. General Semaphore => counting semaphore

Binary Semaphore (Mutex) => locking and unlocking the critical section

1. It will be having the value 1 or 0 => 1 mean available to lock so you can claim the lock once you claim the lock it becomes 0 so that competing threads cannot claim if any of the competing thread try to claim it will be blocked => it has only 1 permit

2. Once a thread release the it permit will be restored to 1 as result competing thread will be unblocked and eventually claim the lock



All the blocked threads will be kept in a FIFO queue

General Semaphore (Counting) => keep track of availability for e.g. in case of producer consumer problem if there multiple slots you can use general semaphore for counting the free slots and number of available items

Two semaphores:

1. For freeslots

2. For number of items available (occupied slots)

‘Semaphore freeSema = new Semaphore(6);

‘Semaphore itemSema = new Semaphore(0);

Consumer => to consume it has to successfully claim() in itemSema semaphore

Initially itemSema has 0 permit therefore if consumer tries to claim() it will be blocked freeSema has 6 permits therefore if producer tries to claim() it will be allowed to claim() if the claim() is successful then permit will be reduced by 1 it will be allowed to produce once production is over producer will call release() itemSema which will increase the permit 0 to 1

After the production

freeSema => 5 permit

itemSema => 1 permit

# FSP

|  |  |
| --- | --- |
| //normal fsp parallel  FAULTY = ( red -> FAULTY| blue -> FAULTY| green -> FAULTY| orange -> kitkat -> FAULTY ).  //buffer  BUFF = ( in[ i : 0..3 ] -> out[ i ] -> BUFF ) .  //constant and range  const MIN\_INT = -3 range INT = 0 .. MAX\_INT  //sum1  const N = 1  range IN = 0..N  range OUT = 0..2\*N  SUM1 = ( in[ a:IN ][ b:IN ] -> TOTAL1[ a + b ] ) ,  TOTAL1[ s : OUT ] = ( out[ s ] -> SUM1 ) .  //sum2  const N = 1  range IN = 0..N  SUM2 = ( in[ **a:IN** ][ **b:IN** ] -> TOTAL2[**a**][**b**] ) ,  TOTAL2[ **x:IN** ][ **y:IN** ] = ( out[**x + y**] -> SUM2 ) .  //params  const N = 5  /\* PARAMS: LL - lower limit, UL - upper limit \*/  PARAMS\_2( LL = 0, UL = N )  = ( in[ i : LL..UL ]  -> add\_2 [i][i + 2]  -> subtract\_1 [i][i - 1]  -> multiply\_4 [i][i \* 4] -> STOP ) . | //if-else process  const ZERO = 0  const MAX\_INT = 3  range INT = ZERO .. MAX\_INT  IS\_ZERO = ( in[ x : INT ] ->  **if** ( x == ZERO )  **then**  ( isZero -> IS\_ZERO )  **else**  ( isNotZero -> IS\_ZERO )) .  //Boolean guard  const TRUE = 1  const FALSE = 0  ( when(TRUE) x -> P | y -> Q ) = ( x -> P | y -> Q )  ( when(FALSE) x -> P | y -> Q ) = ( y -> Q )  ( when(TRUE) x -> P ) = ( x -> P )  ( when(FALSE) x -> P ) = STOP .  //overlapping guard  COUNT ( N = 3 ) = CT[0],  CT[ i : 0..N ] = ( when( i < N ) inc -> CT[i+1] | when( i > 0 ) dec -> CT[i-1] ) .  //concurrent with synchronized  MAKER = ( make -> ready -> MAKER ) .  USER = ( ready -> use -> USER ) .  || MAKER\_USER = ( MAKER || USER ) . |